AJ

PCT WORLD INTELLECTUAL PROPERTY ORGANIZATION
International Bureau

INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(51) International Patent Classification 6 : G06F 9/46 — A1

(11) International Publication Number: WO 98/02812

(43) International Publication Date: 22 January 1998 (22.01.98)

(21) International Application Number: PCT/US97/11886

(22) International Filing Date: 10 July 1997 (10.07.97)

(30) Priority Data:
08/680,266 11 July 1996 (11.07.96) US

(71) Applicant: TANDEM COMPUTERS INCORPORATED [US/US]; 10435 N. Tantau Avenue, LOC. 200-16, Cupertino, CA 95014 (US).

(72) Inventor: SCHOFIELD, Andrew; Lindenbuehl 27, CH-6330 Cham (CH).

(74) Agents: GRANATELLI, Lawrence, W. et al.; Graham & James LLP, 600 Hansen Way, Palo Alto, CA 94304 (US).

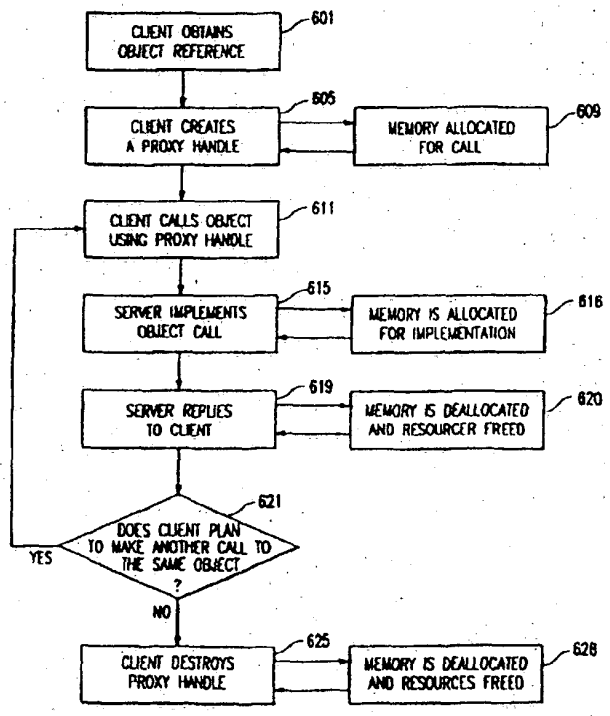
(81) Designated States: JP, European patent (AT, BE, CH, DE, DK, ES, FI, FR, GB, GR, IE, IT, LU, MC, NL, PT, SE).

Published
With international search report.
Before the expiration of the time limit for amending the claims and to be republished in the event of the receipt of amendments.

(54) Title: METHOD AND APPARATUS FOR PERFORMING DISTRIBUTED OBJECT CALLS USING PROXIES AND MEMORY ALLOCATION

(57) Abstract

A method and apparatus for performing distributed object calls uses proxies and memory allocation and deallocation. Specifically, an object reference to an object is obtained. The object reference is used to create a proxy handle data structure that will represent the object. The proxy handle is passed to a client application stub function which calls the object. The stub function is also passed input and output parameters along with exception information. An object request broker finds an appropriate implementation in a server application. The server application allocates memory for implementing the call. The object is implemented and the memory allocated by the server application is deallocated. The server application responds to the client, whereupon the client makes another object call using the same proxy handle or destroys the proxy handle. Multiple initialization of object calls is avoided because the object call can be initialized just once for a particular object. Moreover, resources are preserved by minimizing wild pointers and memory leaks that can occur during the calling and implementation of objects.



FOR THE PURPOSES OF INFORMATION ONLY

Codes used to identify States party to the PCT on the front pages of pamphlets publishing international applications under the PCT.

AL	Albania	**ES**	Spain	**LS**	Lesotho	**SI**	Slovenia
AM	Armenia	**FI**	Finland	**LT**	Lithuania	**SK**	Slovakia
AT	Austria	**FR**	France	**LU**	Luxembourg	**SN**	Senegal
AU	Australia	**GA**	Gabon	**LV**	Latvia	**SZ**	Swaziland
AZ	Azerbaijan	**GB**	United Kingdom	**MC**	Monaco	**TD**	Chad
BA	Bosnia and Herzegovina	**GE**	Georgia	**MD**	Republic of Moldova	**TG**	Togo
BB	Barbados	**GH**	Ghana	**MG**	Madagascar	**TJ**	Tajikistan
BE	Belgium	**GN**	Guinea	**MK**	The former Yugoslav Republic of Macedonia	**TM**	Turkmenistan
BF	Burkina Faso	**GR**	Greece			**TR**	Turkey
BG	Bulgaria	**HU**	Hungary	**ML**	Mali	**TT**	Trinidad and Tobago
BJ	Benin	**IE**	Ireland	**MN**	Mongolia	**UA**	Ukraine
BR	Brazil	**IL**	Israel	**MR**	Mauritania	**UG**	Uganda
BY	Belarus	**IS**	Iceland	**MW**	Malawi	**US**	United States of America
CA	Canada	**IT**	Italy	**MX**	Mexico	**UZ**	Uzbekistan
CF	Central African Republic	**JP**	Japan	**NE**	Niger	**VN**	Viet Nam
CG	Congo	**KE**	Kenya	**NL**	Netherlands	**YU**	Yugoslavia
CH	Switzerland	**KG**	Kyrgyzstan	**NO**	Norway	**ZW**	Zimbabwe
CI	Côte d'Ivoire	**KP**	Democratic People's Republic of Korea	**NZ**	New Zealand		
CM	Cameroon			**PL**	Poland		
CN	China	**KR**	Republic of Korea	**PT**	Portugal		
CU	Cuba	**KZ**	Kazakstan	**RO**	Romania		
CZ	Czech Republic	**LC**	Saint Lucia	**RU**	Russian Federation		
DE	Germany	**LI**	Liechtenstein	**SD**	Sudan		
DK	Denmark	**LK**	Sri Lanka	**SE**	Sweden		
EE	Estonia	**LR**	Liberia	**SG**	Singapore		

METHOD AND APPARATUS FOR PERFORMING DISTRIBUTED OBJECT CALLS USING PROXIES AND MEMORY ALLOCATION

BACKGROUND OF THE INVENTION

1. Field of the Invention

The present invention relates to a method and apparatus for performing distributed object calls using proxies on the client side and memory allocation on the server side. Specifically, the method involves the creation of a proxy handle data structure to be associated with a particular object and its associated object reference. On the client side, object calls to the associated object are then made using the proxy handle, thus allowing multiple calls to the same object and eliminating the need to perform certain initialization functions each time the call is made. On the server side, the server allocates memory in a platform-independent manner.

2. Background

Distributed object computing combines the concepts of distributed computing and object-oriented computing. Distributed computing consists of two or more pieces of software sharing information with each other. These two pieces of software could be running on the same computer or on different computers connected to a common network. Most distributed computing is based on a client/server mode. With the client/server model, two major types of software are used: client software, which requests the information or service, and server software, which provides or implements the information or service.

Object-oriented computing is based upon the object model where pieces of code called "objects"--often abstracted from real objects in the real world--contain data (called "attributes" in object-oriented programming parlance) and may have actions (also known as "operations") performed on it. An object is defined by its interface (or "class" in C++ parlance). The interface defines the characteristics and behavior of a kind of object, including the operations that can be performed on objects of that interface and the parameters to each operation. A specific instance of an object is identified within a distributed object system by a unique identifier called an object reference.

In a distributed object system, a client application sends a request (or "object call")

to a server application. The request contains an indication of the operation to be performed on a specific object, the parameters to that operation, the object reference for that object, and a mechanism to return error information (or "exception information") about the success or failure of a request. The server application receives the request and carries out the request via a server "implementation." The implementation satisfies the client's request for an operation on a specific object. The implementation includes one or more methods, which are the portions of code in the server application that actually do the work requested of the implementation. If the implementation is carried out successfully, the server application may return a response to the client. The server application may also return exception information.

To standardize distributed object systems, the Object Management Group ("OMG"), a consortium of computer software companies, proposed the Common Object Request Broker Architecture ("CORBA"). Under the CORBA standard, an Object Request Broker ("ORB") provides a communication hub for all objects in the system passing the request to the server and returning the response to the client. Commercial ORB's are known in the art and a common type is IBM's System Object Model ("SOM"). On the client side, the ORB handles requests for the invocation of a method and the related selection of servers and methods. When a client application sends a request to the ORB for a method to be performed on an object, the ORB validates the arguments contained in the request against the interface for that object and dispatches the request to the server, starting it if necessary. On the server side, the ORB uses information in the request to determine the best implementation to satisfy the request. This information includes the operation the client is requesting, what type of object the operation is being performed on, and any additional information stored for the request. In addition, the server-side ORB validates each request and its arguments. The ORB is also responsible for transmitting the response back to the client.

Both the client application and the server application must have information about the available interfaces, including the objects and operations that can be performed on those objects. To facilitate the common sharing of interface definitions, OMG proposed the Interface Definition Language ("IDL"). IDL is a definition language (not a programming language) that is used to describe an object's interface; that is, the characteristics and behavior of a kind of object, including the operations that can be performed on those

objects.

IDL is designed to be used in distributed object systems implementing OMG's CORBA Revision 2.0 specification. In a typical system implementing the CORBA specification, interface definitions are written in an IDL-defined source file (also known as a "translation unit"). The source file is compiled by an IDL compiler that generates programming-language-specific files, including client stub files, server stub files, and header files. Client stub files are language-specific mappings of IDL operation definitions for an object type into procedural routines, one for each operation. When compiled by a language-specific compiler and linked into a client application, the stub routines may be called by the client application to invoke a request. Similarly, the server stub files are language-specific mappings of IDL operation definitions for an object type (defined by an interface) into procedural routines. When compiled and linked into a server application, the server application can call these routines when a corresponding request arrives. Header files are compiled and linked into client and server applications and are used to define common data types and structures.

In a system implementing the CORBA specification, object calls are made by calling the appropriate client stub function. The parameters to these stub functions typically include the input parameters for the requested operation and the object reference of the object. The CORBA specification, however, has certain performance-related drawbacks. First, each time an object is called, certain configuration operations must be performed. For instance, the object must be validated to ensure its existence. In addition, memory (or another resource) is often allocated for the call. The interface must be determined. Finally, transport of the call is prepared (e.g., opening files, preparing sockets, etc...). Performing each of these operations prior to each object call can be very time-consuming. In a time-critical application, the overhead associated with each object call can significantly affect performance.

Further, on the server side, performance is affected by the improper allocation and deallocation of memory ("garbage collection"). In many distributed object systems, memory can be allocated for the implementation of an object call. Once the implementation has been performed and the server responds to the call, however, memory is not automatically deallocated. This failure to deallocate memory leads to wasted resources and, therefore, slower performances. In addition, the failure to remove pointers or addresses

pointed to by pointers can lead to larger problems, such as system errors and application shutdowns.

Certain systems have attempted to use garbage collection in the implementation of object calls. Those systems, however, use platform-specific methods for deallocating memory. Thus, code portability is often sacrificed in favor of proper memory allocation and deallocation.

Accordingly, a need exists for a method for improving object call performance by eliminating the need to configure an object call each time the same object is called.

Further, a need exists for a platform-independent method for improving object call performance by allocating and deallocating memory during object implementation.

SUMMARY OF THE INVENTION

The present invention provides a method and apparatus for performing object calls that improves performance by eliminating the need to configure an object call each time the same object is called. The present invention also improves object call performance by allocating and deallocating memory during object implementation.

More particularly, the present invention is directed to a method for calling an object via a generated stub function within a client application to a server application. The call is performed by first obtaining an object reference that refers to the object. Next, the object reference is used to create a proxy handle to represent the object in calls to the object. The object is then called via the stub function. The stub function is passed the proxy handle, an input parameter of an operation to be performed on the object, an output parameter of the operation, and exception information. The object is then implemented via a method function in the server application. The server application responds to the caller and completes the call. The proxy handle may be used again for a subsequent call to the same object. Once the final object call has been made, the proxy handle is destroyed.

The creation of the proxy handle automatically allocates resources for the object call. When the proxy handle is destroyed, those resources are freed. Moreover, since the proxy handle can be used for multiple calls to the same object, the client application is not required to perform any initialization that may have been performed during the previous object calls. For instance, object validation, file opening, and socket preparation are events that are commonly performed during each object call. These initialization events may be placed in a

time-critical portion of the application to save resources. Finally, multiple proxy handles may be used for the same object to force users to comply with certain requirements before implementing a particular feature of an object. In this manner, files and similar objects may be pseudo-encrypted.

In an alternative embodiment, resources may be allocated and deallocated during the actual implementation of the object. In this embodiment, the method function in the server application calls a function to allocate the appropriate resources. The function returns a pointer to memory that has been allocated on the heap in the server computer. Upon completion of the method function, the memory is automatically deallocated. Automatic deallocation of memory reduces the likelihood of random pointers and memory leaks that can often occur during the implementation of an object.

A more complete understanding of the method and apparatus for performing object calls will be afforded to those skilled in the art, as well as a realization of additional advantages and objects thereof, by a consideration of the following detailed description of the preferred embodiment. Reference will be made to the appended sheets of drawings which will first be described briefly.

BRIEF DESCRIPTION OF THE DRAWINGS

Fig. 1 is a block diagram of a client/server computing system utilizing the method of the present invention.

Figs. 2a and 2b are diagrams of alternative Common Execution Environment capsule configurations.

Fig. 3 is a diagram of a Common Execution Environment capsule and its core components.

Fig. 4 is a diagram of the compilation and linking of IDL source code into client and server applications.

Fig. 5 is a flow chart describing the steps involved in the method of the present invention.

Fig. 6 is a flow chart describing the steps involved in an alternative embodiment of the method of the present invention.

Fig. 7 is a diagram showing a PIF data structure.

Fig. 8 is a diagram showing an entry data structure.

Fig. 9 is a diagram showing an operation data structure.

Fig. 10 is a diagram showing a union data structure.

DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENT

Reference will now be made in detail to the preferred embodiments of the invention, examples of which are illustrated in the accompanying drawings. Wherever possible, the same reference numbers will be used throughout the drawings to refer to the same or like parts.

I. System Overview

As illustrated in Figure 1, the method of the present invention is designed for use in a distributed (client/server) computing environment 10. The client and server systems are connected by network connections 12, such as internet connections or the connections of a local area network. The server computer 11 communicates over a bus of I/O channel 20 with an associated disk storage subsystem 13. The server system 11 includes a CPU 15 and a memory 17 for storing current state information about program execution. A portion of the memory 17 is dedicated to storing the states and variables associated with each function of the program which is currently executing on the client computer. The client computer 21 similarly includes a CPU 27 and associated disk memory 23, and an input device 29, such as a keyboard or a mouse and a video display terminal ("VDT") 33. The client CPU communicates over a bus or I/O channel 40 with a disk storage subsystem 33 and via I/O channel 41 with the keyboard 29, VDT 33 and mouse 31. Both computers are capable of reading various types of media, including floppy disks and CD-ROMs.

The client memory 27 includes a client application 77 and client stubs 79 loaded therein. Similarly, the server memory 17 includes a server application 87 and server stubs 89. In addition, both the client memory and the server memory include an execution environment 75, 85 (as discussed below).

The client/server model as shown in Figure 1 is merely demonstrative of a typical client/server system. Within the context of the present invention, the "client" is an application that requests that operations be performed on an object while the "server" is an application that implements the operation on the object. Indeed, both the client and server application may reside on the same computer and within a common capsule, as discussed

below. Most likely, however, the client and server application will reside on separate computers using different operating systems. The method of the present invention will be discussed with reference to two capsules running on separate machines.

The method and apparatus of the present invention may be utilized within any distributed computing environment. In a preferred embodiment, the Common Execution Environment ("CEE"), which is a component of the Tandem Message Switching Facility ("MSF") Architecture is used. The CEE activates and deactivates objects and is used to pass messages between client and server applications loaded in CEE capsules. The CEE may be stored in the memory of a single machine. More likely, however, the CEE and client and server applications will be loaded on multiple machines across a network as shown in Figure 1. The client-side CEE 75 is stored in the client memory 27. The server-side CEE 80 is stored in server memory 17.

The CEE uses a "capsule" infrastructure. A capsule encapsulates memory space and execution stream. A capsule may be implemented differently on different systems depending upon the operating system used by the system. For instance, on certain systems, a capsule may be implemented as a process. On other systems, the capsule may be implemented as a thread. Moreover, client and server applications may be configured within different capsules contained on different machines as shown in Figure 1. Alternatively, the different capsules may be configured as shown in Figure 2. Figure 2a shows a client application 79 loaded in a single capsule and a server application 89 may be loaded in a separate capsule 85. Both capsules, however, are stored on the same machine. Both the client and server applications may also be loaded within a single capsule 81 on the same machine 21 as shown in Figure 2b. As stated above, the method of the present invention will be described with reference to the multiple capsule, multiple machine case. Accordingly, the client 12 and server machine 11 include a client-side CEE 75 and a server-side CEE 85 loaded in their respective memories.

Figure 3 shows a CEE capsule 70 contained in a client computer memory 27 (not shown) that includes the CEE 75 and certain of the core CEE components and implementations of objects contained within Implementation Libraries 71. The Implementation Libraries 71 include the client application 79 (or the server application in the case of the server capsule) and client stubs 77 generated from the IDL specification of the object's interface, as described below. The Implementation Libraries 71 and the CEE 75

interact through the down-calling of dynamically-accessible routines supplied by the CEE and the up-calling of routines contained in the Implementation Library. The CEE 75 can also receive object calls 82 from other capsules within the same machine and requests 84 from other CEE's. The client-side CEE 75 and the server-side CEE 85 may communicate using any known networking protocol.

Objects implemented in a CEE capsule may be configured or dynamic. Configured objects have their implementation details stored in a repository (such as the MSF Warehouse 85) or in initialization scripts. Given a request for a specific object reference, the CEE 75 starts the appropriate capsule based on this configuration data. The capsule uses the configuration data to determine which Implementation Library to load and which object initialization routine to call. The object initialization routine then creates the object. Dynamic objects are created and destroyed dynamically within the same capsule. Dynamic objects lack repository-stored or scripted configuration information.

The following paragraphs describe a system-level view of how the Implementation Libraries interacts with the CEE 75. The CEE 75 implements requests to activate and deactivate objects within a capsule. In addition, the CEE facilitates inter-capsule object calls 83 as well as requests from other CEE's 87, as discussed above. Object activation requests arise when an object call from a client or server application must be satisfied. To activate an object, the CEE 75 loads the appropriate Implementation Library (if not already loaded) containing the object's methods and then calls a configured object initialization routine. The initialization routine specifies which interface it supports and registers the entry points of the object's methods to be called by the CEE at a later time.

When the client and server systems start, the CEE runs its own initialization. This initialization tells client and server CEE's where to locate the various Implementation Libraries. Once located by the CEE, the initialization routines in the client and server applications are called. The initialization routines contained in the client and server applications must first carry out any required application-specific initialization. Next, both the client and server initialization routines down-call a CEE function (contained in a dynamic library as stated above) called CEE_INTERFACE_CREATE to specify the object's interface. The interface description is normally generated from an IDL description of the interface contained in an IDL source file, as discussed below. CEE_INTERFACE_CREATE creates an interface and returns a handle to the newly created

interface. The handle is a unique identifier that specifies the newly-created interface. The server application initialization routine then uses the interface handle to down-call CEE_IMPLEMENTATION_CREATE. CEE_IMPLEMENTATION_CREATE creates an implementation description that can be used by one or more objects. CEE_IMPLEMENTATION_CREATE returns an implementation handle that is a unique identifier specifying the implementation for each operation in the interface. Finally, the server application initialization routine uses the implementation handle to call CEE_SET_METHOD and specifies the actual addresses of specific method routines of the implementation as contained in the server application. The CEE then has sufficient information to connect object calls in the client application to specific methods in the server application.

Figure 4 shows how IDL source files are compiled and linked into client and server applications that will utilize the method and apparatus of the present invention. First, an IDL source file 101 is prepared containing IDL interface definitions. An IDL compiler 103 compiles the source file 101. The IDL compiler 103 parses the code 101 to produce a Pickled IDL file ("PIF") file 105 for storage of the compiled source code. A code generator 111 then parses the PIF file. The generation of a PIF file is described below in Section IV. The code generator 111 generates files in the language of the client and server applications. If the client and server applications are in different languages, different code generators are used. Preferably, the code generator 111 and IDL compiler 103 may be combined in a single application to produce language-specific code. The code generator 111 produces a client stub file 115 containing client stub functions and a server stub file 117 containing definitions for object implementations. The client stub functions include synchronous and asynchronous calls to the CEE. The client stub file 115 and the server stub file 117 are compiled by programming language-specific compilers 121, 123 to produce compiled client stub object code and compiled server stub object code. Similarly, a client application 79 and a server application 89 are compiled by programming-language-specific compilers to produce compiled client application object code and compiled server application object code. The client application 121 and the server application 123 also include a header file 119 generated by the code generator 111. The header file 119 contains common definitions and declarations. Finally, the compiler 121 links the client application object code and the client stub object code to produce an implementation library 71.

Similarly, a second compiler links the server application object code server stub object code to produce another implementation library 81.

II. Proxy Creation

Now, with reference to Figures 5 and 6, the method of the present invention will be described. The method of the present invention will be described with reference to a client application executing in a capsule on a client computer and a server application executing in a separate capsule on a server computer. Figure 5 is a flow chart depicting the steps involved in utilizing a proxy handle to call an object. In a first step 501, an object reference for the desired object must be obtained. The object reference may be obtained in a number of ways. Client applications usually receive the reference from configuration data, directories or invocations on other objects to which they have object references. An object reference can be converted to a string name that can be stored in files to be accessed later.

Once the object reference has been obtained, the object call may be performed. In the method of the present invention, the object call is performed by first obtaining a "proxy handle" to the object reference. The proxy handle is a unique identifying data structure (a "proxy object") for a particular object reference. The proxy structure contains information about an object and calls to that object. Calls can be made to the specified object using the proxy handle. The proxy handle facilitates calls to the same object and prevents overhead that occurs in multiple calls to the same object. By creating a proxy handle via which object calls can be made, certain initialization routines may be performed once through the proxy handle while allowing multiple calls to the proxied object. In addition, the proxy handle facilitates the use of asynchronous calls to an object (discussed below). In a preferred embodiment, a proxy handle is created in step 503 by down-calling the client-side CEE function, CEE_PROXY_CREATE, in the client application. That function is defined in the C programming language as follows:

```
CEE_PROXY_CREATE (
        const char      *objref,
        const char      *intf_handle,
        char            *proxy_handle);
```

The function receives the object reference, *objref*, and an interface handle, *intf_handle*, and

returns a proxy handle identifying the newly created proxy object. As discussed above, in connection with Figure 3, an interface must be created for each object. An interface defines the operations available for a collection of similar objects. The interface is defined in IDL and compiled and linked into a client application. The client application calls CEE_INTERFACE_CREATE in its initialization routine to specify the interface. The client-side CEE returns an interface handle that can be used to create any number of objects or proxies.

In a preferred embodiment of the present invention, the proxy object is represented by a structure containing the following fields:

link;
call_link;
self;
nor;
state;
call_active;
destroy;
lock_count;
**intf*;
call_compl_rtn;
call_compl_tag1;
call_complt_tag2;
call_compl_sts;
operation_idx;
**client_allocated_params*;
**server_allocated_params*;
**obj*;
operation_idx_table;
max_response_size;
**req_area*;
**rsp_area*;
**rsp_param_buf*;
ochan;

Each of the components of the proxy object data structure will now be discussed. The addresses and values stored in each of these components is modified by the client-side CEE with each call to the object referred to by the proxy structure. The client-side CEE maintains a linked list of proxy structures. The *link* member of each proxy structure contains a pointer to the next entry in this list of proxy structures.

In a preferred embodiment, object calls can be either synchronous (client application requests that an operation be performed on an object and waits for a response) or asynchronous (client application requests that an operation be performed on an object and continues to do other work). When asynchronous object calls are dispatched in the same capsule, each call is queued onto a linked list contained in an object structure that exists for every object when activated. The *call_link* parameter is a link to the list of calls. The *self* member is the handle of this particular proxy structure. This handle is returned to the client during CEE_PROXY_CREATE.

The object reference passed to CEE_PROXY_CREATE is stored in the *nor* member. The *state* parameter indicates whether the proxy structure includes a pointer to an internal object structure, an external object (via a client port), or a non-existing object structure (is stale). If the proxy is internal, a pointer to the object is contained in the *obj* parameter. If the object is in a different capsule, the *ochan* parameter contains a pointer to a client port handle or other information required to communicate with the object.

The *call_active* member holds a true/false value. The *call_active* member is set to true if an object call is outstanding for this particular proxy handle. Only one object call can be outstanding on a given proxy. The *lock_count* member is incremented to prevent the proxy structure from being destroyed. It is decremented when the structure is no longer needed. The *destroy* member is a true/false value that is set to true if this proxy structure should be destroyed when *lock_count* drops to zero. The *intf* member is the address of the *intf* structure that describes the interface (discussed above).

The next four structure members, *call_compl_rtn*, *call_compl_tag1*, *call_compl_tag2*, *call_compl_sts*, are used to implement asynchronous object calls. Asynchronous calls to an object in a server application are made by passing the address of a completion routine to the client stub function when called. The client stub function, in turn, calls the client-side CEE and provides the completion function address. The client-side CEE

stores the completion function address in the *proxy* structure upon creation of the proxy handle. When the object call completes, the client-side CEE calls the completion routine specified in the *proxy* handle. The routine is called to notify the client application that the call has completed. While the object is being implemented, the client application can continue performing other functions. The member *call_compl_rtn* contains the address of the completion routine. Since multiple calls may be made to the same proxied object, the client application can identify the call by using the *call_compl_tag1* parameter when the object call is made. The *call_compl_tag1* identifier is passed to the client stub function. These identifiers are specified in the proxy structure by the members *call_compl_tag1* and *call_compl_tag2*. The *call_compl_sts* indicates the call completion status for asynchronous calls that could not be called.

The *operation_idx* member specifies which of the object's operations is to be called. Operation identifiers are generated by the code generator for each operation in the interface. The *allocated_params* member is a pointer to the parent of temporarily allocated parameters (used for unbounded types and the like). The deallocation of this member performs garbage collection on the next call to the object referenced by this proxy structure. The *operation_idx_table* parameter is a pointer to an operation index translation table that is used only if the object is contained in the same capsule.

Memory allocation is performed utilizing the *max_response_size*, *req_area*, *rsp_area*, and *rsp_param_buf* members. The *rsp_param_buf* member points to a buffer containing the response parameters. The next time that this proxy object is used, the buffer will be deallocated. The *max_response_size* member is the maximum expected response size. This is used to allocate the *rsp_area* member. The *req_area* member points to an *area* structure that will be used for the request. The *rsp_area* points to an *area* structure that will be used for the response to the object call. The *area* structure contains the following fields, as defined in C:

desc;

**data*;

curlen;

The *area* structure contains an object call area descriptor and a pointer to data and the current length of that data.

The call by the client application to CEE_PROXY_CREATE causes the client-side CEE to automatically allocate memory for the object call in step 508. Memory in the client computer is allocated along with any additional resources necessary for making the call. Once the proxy handle is destroyed (through a down-call to CEE_PROXY_DESTROY, discussed below), the memory and any allocated resources are freed. Memory allocated for variable-sized output parameters from an object call are deallocated when the next object call is made using the same proxy handle.

The proxy handle is used to make all subsequent calls to the object referred to by the proxy. The object call is made in step 511 by calling the appropriate stub function in the client application and passing the proxy handle and input and output parameters along with exception information to the function. The stub function, in turn, down-calls CEE_OBJECT_CALL, defined in C as follows:

```
CEE_OBJECT_CALL (
        const char      *proxy_handle,
        long            operation_idx,
        void            **param_vector);
```

The proxy handle is specified by the *proxy handle* parameter. The parameter *operation_idx* specifies which of the object's methods is to be called. This parameter is an index to the required method in the interface description that was supplied when the interface was created. Finally, the *param_vector* parameter is an array of pointers to the object call's input parameters, output parameters, and exception structure. The address of the exception structure is the first element in the array. If the operation is not of type *void*, then the following element contains the address of the variable to receive the operation's result. Subsequent elements contain the addresses of the operation's input and output parameters in the same order as they were defined in IDL.

The call is then transported to the server using any transport mechanism. In step 515, the server application implements the object. This is performed by the server-side CEE which up-calls the appropriate method routine. The method routine is passed the *param_vector* parameter containing the addresses of all the input and output parameters. When the method exits, a response is sent to the caller in step 519 and the object call is complete.

Once the first call has completed, the proxy handle may be used again to make further calls to the same object. Each subsequent call to the object may be made without validating the object or performing other start-up operations. Thus, the proxy creation step can be placed in a non-time-critical portion of the client application and object calls can be made in a time-critical portion of the application.

Following the final object call for a specified proxy handle, the proxy handle is destroyed in step 521. This is accomplished by calling CEE_PROXY_DESTROY in the client application, defined in C as follows:

```
CEE_PROXY_DESTROY (
        const char          *proxy_handle);
```

The proxy handle is passed to the function. The client-side CEE destroys the proxy handle and frees all previously-allocated resources for the proxy handle in step 528. Alternatively, an object call may be canceled and all of the resources associated with the call may be deallocated by destroying the proxy while the call is outstanding.

III. Proxy Creation And Memory Allocation

Figure 6 shows an alternative embodiment of the object call method of the present invention. In this embodiment, memory is allocated during the implementation of the object as well as during the object call.

Steps 601-611 are similar to the steps described above. Thus, in step 601, an object reference is obtained. Next, a proxy handle is obtained by down-calling CEE_PROXY_CREATE which returns the proxy handle. The object call is then made in step 611 using CEE_OBJECT_CALL, which is passed the proxy handle of the referenced object.

In step 613, the server-side CEE up-calls the appropriate method routine in the server application. The method routine, in step 616 when called, down-calls a server-side CEE function to allocate memory. That function, CEE_TMP_ALLOCATE is defined in C as follows:

```
CEE_TMP_ALLOCATE (
        const char          *call_id,
        long                len,
```

void **ptr);

The function uses the *call_id* parameter to track a particular object call. Each call to the object is given a unique *call_id* by the server application. Thus, once the call is made, the server implementation provides an id for the call in the *call_id* parameter. The number of bytes to allocate is specified in the *len* parameter. The function returns the address of the allocated memory through the *ptr* parameter.

The object's method is performed by the server application in step 619. The server application responds to the caller in step 615. Upon exiting the method function, the memory allocated under the down-call to CEE_TMP_ALLOCATE is freed in step 620. The client application then makes another object call in step 611 or destroys the proxy handle in step 625. If the proxy handle is destroyed, the memory allocated in step 609 is automatically deallocated by the client-side CEE in step 628.

Memory can be prematurely deallocated using CEE_TMP_DEALLOCATE. That function is defined as:

CEE_TMP_DEALLOCATE (
In void *ptr);

The function is passed the *ptr* parameter that was provided by CEE_TMP_ALLOCATE. The CEE frees the address pointed to by that parameter.

IV. Creating a Pickled IDL Format Data Structure

The Pickled IDL Format ("PIF") data structure is designed to be used in conjunction with IDL compilers and code generators loaded in the client memory 23 and server memory 17. The data structure is based upon an IDL source file stored in memory 23 or in memory 17. The source file may also be contained on a computer-readable medium, such as a disk. The data structure of the present structure contains a parse tree representing the IDL source file. The data structure can be stored in memory 23 or in memory 17 or on a computer-readable medium, such as a disk. The data structure that represents the source file is referred to as a Pickled IDL Format ("PIF"). The PIF file can be accessed at run-time by clients and servers that use the interfaces defined in the source file. The parse tree contained in the PIF file is an array using array indices rather than pointers. The use of array indices

permits the resulting parse tree to be language-independent. The first element of the array is unused. The second element of the array (index 1) is the root of the parse tree that acts as an entry point to the rest of the parse tree.

The data structure, *tu* 701, is shown in Figure 7, and defined in IDL as follows:

```
struct tu_def {
        sequence<entry_def>     entry;
        sequence<string>        source;
}
```

The data structure 701 contains a sequence (a variable-sized array) of parse tree nodes 705, each of type entry_def (defined below) and a sequence of source file lines 707. The sequence of source file lines 707 is a sequence of strings containing the actual source code lines from the IDL source file.

Each parse tree node (or "entry") 705 consists of a fixed part containing the name of the node and its properties as well as a variable portion that depends upon the node's type. The parse tree node is shown in Figure 8 and defined in IDL as follows:

```
struct entry_def {
        unsigned long entry_index;
        string          name;
        string          file_name;
        unsigned long line_nr;
        boolean         in_main_file;
        union u_tag switch (entry_type_def) {
                case entry_argument: argument_def argument_entry;
                case entry_array: array_def array_entry;
                case entry_attr: attr_def attr_entry;
                case entry_const:  const_def const_entry;
                case entry_enum: enum_def enum_entry;
                case entry_enum_val: enum_val_def enum_val_entry;
                case entry_except: except_def except_def_entry;
                case entry_field: field_def field_def entry;
                case entry_interface: interface_def interface_entry;
```

```
                case entry_interface_fwd: interface_fwd_def interface_fwd_entry;
                case entry_module: module_def module_entry;
                case entry_op: op_def op_entry;
                case entry_pre_defined: pre_defined_def pre_defined_entry;
                case entry_sequence: sequence_def sequence_entry;
                case entry_string: string_def string_entry;
                case entry_struct: struct_def struct_entry;
                case entry_typedef: typedef_def typedef_entry;
                case entry_union: union_def union_entry;
                case entry_union_branch: union_branch_def union_branch_entry;
        } u;
};
```

The fixed part of the parse tree node includes *entry_index* 805, an unsigned long which is the index for this particular entry in the parse tree. The unqualified name of the entry is contained in the field *name* 807. The name of the original IDL source file is contained in the field *file_name* 811. The field line_nr 813 contains the line number in the IDL source file that caused this parse tree node to be created. The boolean *in_main_file* 815 indicates whether or not the entry is made in the IDL source file specified on the command line or whether the entry is part of an "include" file. Following these fields, the parse tree node includes a variable portion--a union 817 having a discriminator, *entry_type_def*. The union discriminator, *entry_type_def*, specifies the type of node and which variant within *entry_def* is active. *Entry_type_def* is an enumeration defined as follows:

```
enum entry_type_def {
        entry_unused,
        entry_module,
        entry_interface,
        entry_interface_Fwd,
        entry_const,
        entry_except,
        entry_attr,
```

```
        entry_op,
        entry_argument,
        entry_union,
        entry_union_branch,
        entry_struct,
        entry_field,
        entry_enum,
        entry_enum_val,
        entry_string,
        entry_array,
        entry_sequence,
        entry_typedef,
        entry_pre_defined
    };
```

Entry_type_def includes a list of the various types of parse tree entries. Each parse tree entry represents a constant integer that is used in the switch statement contained in *entry_def*. For each entry, the union *u_tag* will include a different type of structure. The first enumerated value entry_unused corresponds to the value zero and is not used in determining the type of the union.

If the parse tree entry is a module (specified by the value entry_module) the variable portion of the parse tree entry is a data structure including a sequence of module definitions. Each module definition is an unsigned long acting as an index in the parse tree array.

If the parse tree entry is an interface, as specified by the value entry_interface, the variable portion of the parse tree is a data structure including a sequence of local definitions and a sequence of base interfaces from which this interface inherits. If the parse tree entry is a forward declaration of an interface (entry_interface_fwd), the union is an unsigned long containing the index of the full definition.

Constants (entry_const) are represented in a parse tree node as a structure containing the value of the constant. A union and switch/case statement are preferably used to discriminate between the various base type constants (boolean constant, char constant, double constant, etc...) that may be included in the source file.

Exceptions (entry_except) are represented in a parse tree node as a structure containing a sequence of fields. An attributes (entry_attr) is represented as a data structure containing a boolean value that indicates whether the attribute is read-only and an unsigned long that indicates the data type.

If the parse tree entry is an operation (op_def), the variable portion 817 of the entry data structure 905 is a data structure as shown in Figure 9. The data structure 817 contains a boolean 905 that indicates whether or not the operation has a one-way attribute, an unsigned long 907 that indicates the return type, a sequence of arguments 909 to the operation, a sequence of exceptions 911 to the operation, and a sequence of strings 1313 that specify any context included in the operation. If the parse tree entry is an argument to a particular operation (entry_argument), the variable portion of the parse tree entry is a structure containing unsigned longs that indicate the data type and direction of the argument.

If the parse tree entry is a union (entry_union), it is represented in the parse tree entry as shown in Figure 10. The data structure 817 contains an unsigned long specifying the discriminator 1003 and an unsigned long specifying the type 1005. The type is preferably specified using an enumerated list of base types. The structure 817 further includes a sequence of the union's fields 1007. If the parse tree entry is a union branch (entry_branch), the variable portion of the parse tree entry is a structure containing an unsigned long indicating the base type of the branch, a boolean indicating whether or not the branch includes a case label, and the value of the discriminator. Since the value is of a particular data type, preferably an enumerated list of the various base types is used to specify the value within the structure used to represent the union branch.

For data structures (entry_struct), the variable portion of the parse tree entry includes a structure containing a sequence of the specified structure's fields. Enumerated values (entry_enum) are represented by a structure containing a sequence of enumerated values. Enumerations of an enumerated type (entry_enum_val) are represented in the parse tree entry by a structure containing an unsigned long holding the enumeration's numerical value.

If the parse tree entry is a string (entry_string), the variable portion of the parse tree entry is a structure containing the string's maximum size. A maximum size of zero implies an unbounded string. An array (entry_array) is represented in the parse tree entry by a structure containing an unsigned long holding the array's base type and a sequence of longs

holding the array's dimensions. A sequence (entry_sequence) is represented by a structure containing unsigned longs holding the sequence's base type and the sequence's maximum size.

For type definitions (entry_typedef), the parse tree entry includes a structure containing an unsigned long value indicating the type definition's base type. Predefined types (entry_pre_defined) are represented by a structure containing the data type. To specify the type, preferably an enumeration of the various base types are used.

Once the IDL source file has been described using the *tu* data structure, the data structure may be transported to a file or database using any known methods.

Having thus described a preferred embodiment of a method for making object calls using proxies and memory allocation/deallocation, it should be apparent to those skilled in the art that certain advantages of the within system have been achieved. It should also be appreciated that various modifications, adaptations, and alternative embodiments thereof may be made within the scope and spirit of the present invention. The invention is further defined by the following claims.

CLAIMS

What is Claimed is:

1. A method for calling an object stored in a computer memory, the method comprising the steps of:

obtaining an object reference that refers to the object, the object reference being obtained by a client application;

creating a proxy handle data structure to represent the object, the data structure containing the object reference and, if the object is within a same process as the client application, a pointer to the object and, if the object is in a different process from the client application, a pointer to information required to communicate with the object;

requesting that an operation be performed on the object via a stub function in the client application, the stub function being passed the proxy handle data structure and an input parameter of the operation;

implementing the operation on the object in a server application; and

responding to the client application.

2. The method for calling an object, as recited in Claim 1, further comprising the step of destroying the proxy handle after the call has completed.

3. The method for calling an object, as recited in Claim 1, further comprising the step of destroying the proxy handle while the call is outstanding.

4. The method for calling an object, as recited in Claim 2, wherein the object call is performed using an execution environment accessible by the client application.

5. The method for calling an object, as recited in Claim 5, wherein the step of creating the proxy handle is performed by the execution environment.

6. The method for calling an object, as recited in Claim 6, wherein the step of destroying the proxy handle is performed by the execution environment.

7. The method for calling an object, as recited in Claim 1, wherein the stub function is also passed an output parameter of the operation to be performed on the object.

8. The method for calling an object, as recited in Claim 7, wherein the stub function is also passed exception information.

9. The method for calling an object, as recited in Claim 1, further comprising the steps of:

allocating a block of memory in the computer memory for implementing the operation; and

deallocating the block of memory in the computer memory following the implementation of the operation.

10. The method for calling an object, as recited in Claim 9, wherein the block of memory is allocated by an execution environment accessible to the server application.

11. The method for calling an object, as recited in Claim 1, wherein the client application is stored in the computer memory.

12. The method for calling an object, as recited in Claim 1, wherein the client application is stored in a second computer memory.

13. A method for calling an object stored in a computer memory, the method comprising the steps of:

obtaining an object reference to refer to the object, the object reference being obtained by a client application;

creating a proxy handle data structure to represent the object, the data structure containing the object reference and, if the object is within a same process as the client application, a pointer to the object and, if the object is in a different process from the client application, a pointer to information required to communicate with the object

requesting that an operation be performed on the object via a stub function in the

client application, the stub function being passed the proxy handle, and an input parameter of the operation;

allocating a block of memory in the computer memory to implement the operation on the object;

implementing the operation on the object in the server application;

deallocating the block of memory in the computer;

responding to the client application; and

destroying the proxy handle.

14. The method for calling an object, as recited in Claim 13, wherein the step of creating a proxy handle data structure is performed by an execution environment accessible to the client application.

15. The method for calling an object, as recited in Claim 14, wherein the step of destroying the proxy handle is performed by the execution environment.

16. The method for calling an object, as recited in Claim 13, wherein the stub function is also passed an output parameter of the operation to be performed on the object.

17. The method for calling an object, as recited in Claim 16, wherein the stub function is also passed exception information.

18. A computer program product, comprising:

a computer useable medium having computer readable code means embodied therein for performing an object call, the computer readable program code means comprising:

software means for obtaining an object reference that refers to the object, the object reference being obtained from within a client application;

software means for creating a proxy handle data structure to represent the object, the data structure containing the object reference and, if the object is within a same process as the client application, a pointer to the object and, if the object is in a different process from the client application, a pointer to information required to communicate with the object;

software means for requesting that an operation be performed on the object via a stub

function in the client application, the stub function being passed the proxy handle data structure and an input parameter of the operation;

software means for implementing the operation on the object in a server application; and

software means for responding to the client application.

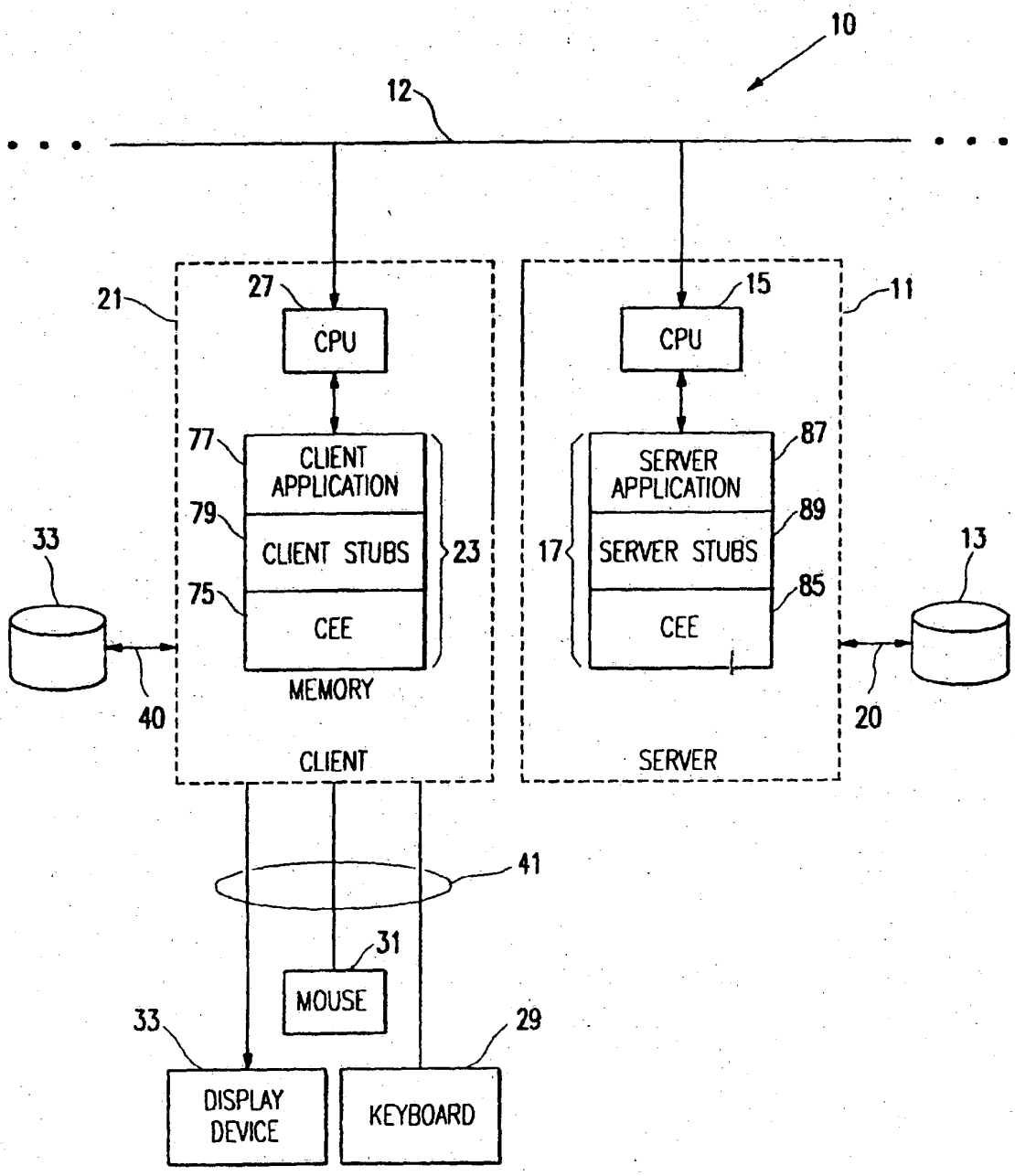

10
12
21
27
CPU
77
CLIENT APPLICATION
79
CLIENT STUBS
75
CEE
23
MEMORY
33
40
CLIENT
15
CPU
11
87
SERVER APPLICATION
89
SERVER STUBS
85
CEE
17
13
20
SERVER
41
31
MOUSE
33
29
DISPLAY DEVICE
KEYBOARD

FIG. 1

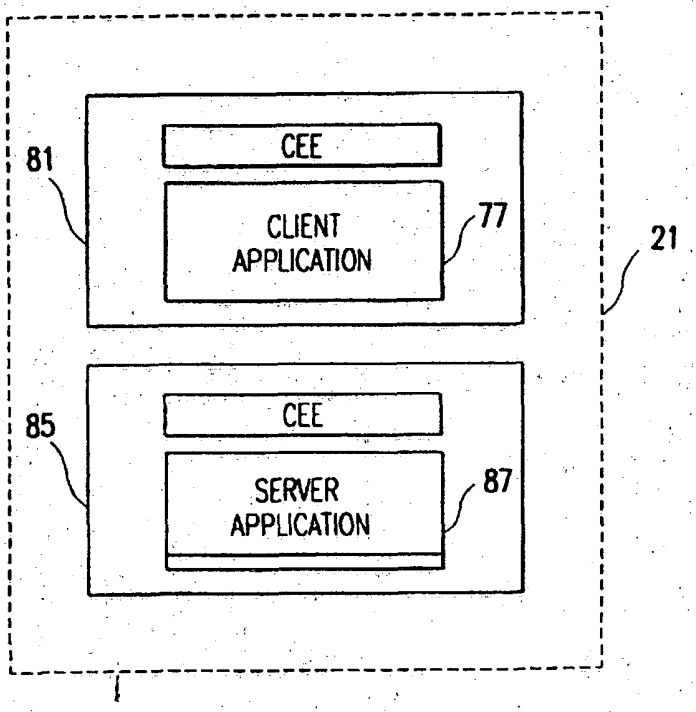

81
CEE
CLIENT APPLICATION
77
21
85
CEE
SERVER APPLICATION
87

FIG. 2A

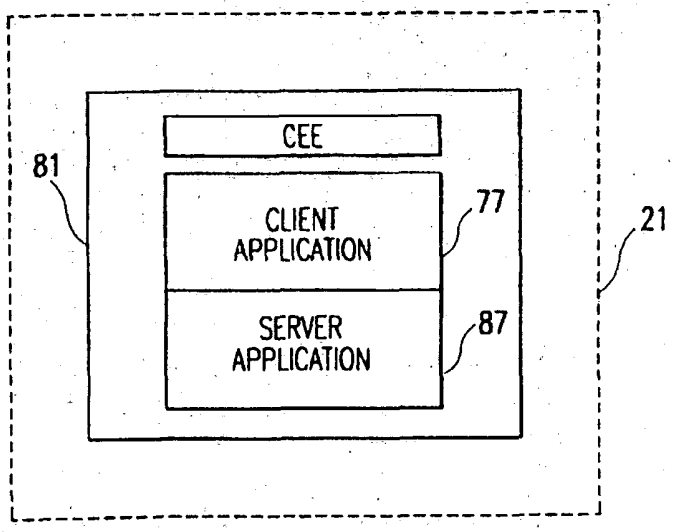

81
CEE
CLIENT APPLICATION
77
21
SERVER APPLICATION
87

FIG. 2B

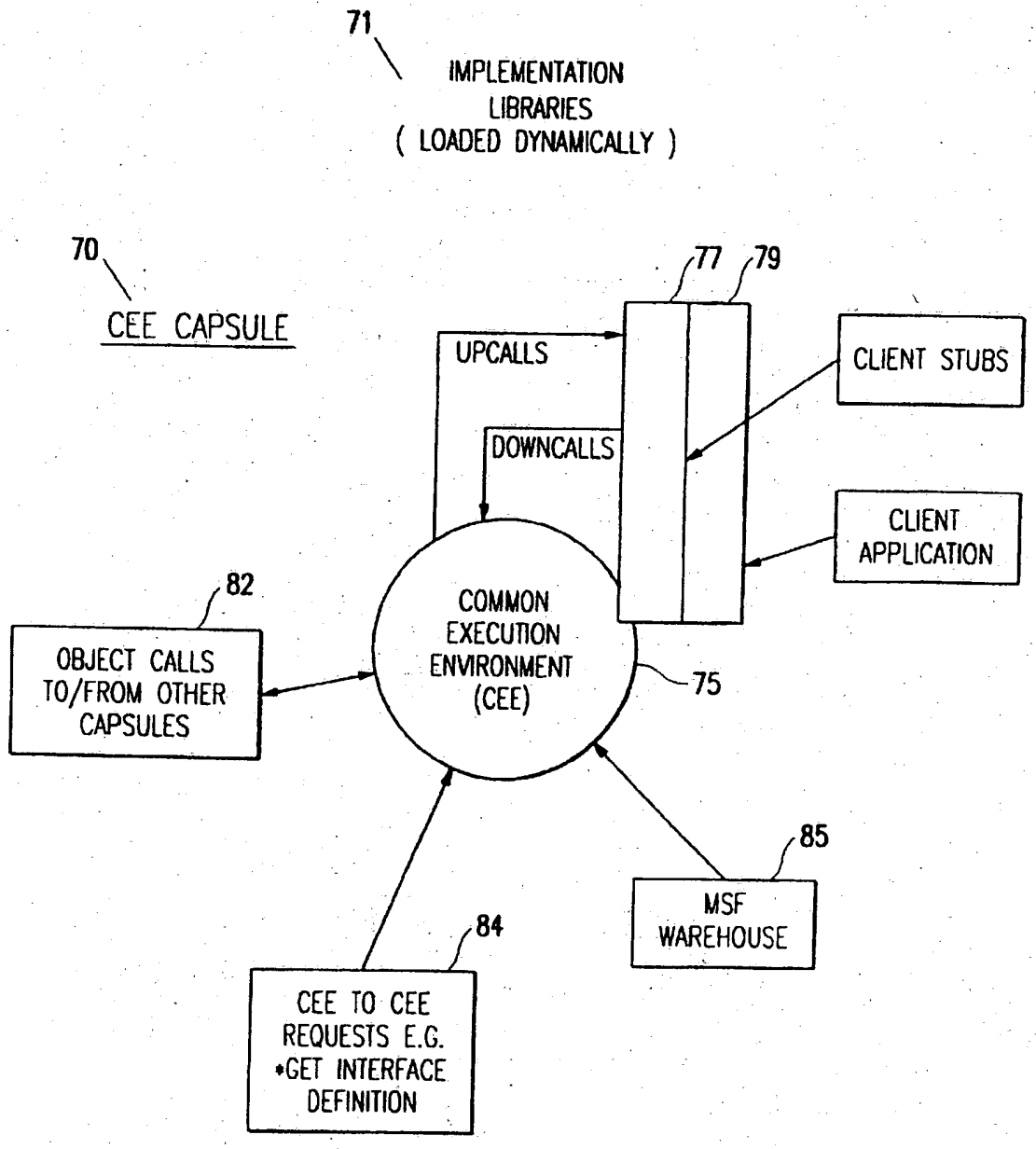

71
IMPLEMENTATION LIBRARIES (LOADED DYNAMICALLY)
70
CEE CAPSULE
77
79
UPCALLS
DOWNCALLS
CLIENT STUBS
CLIENT APPLICATION
82
OBJECT CALLS TO/FROM OTHER CAPSULES
COMMON EXECUTION ENVIRONMENT (CEE)
75
85
MSF WAREHOUSE
84
CEE TO CEE REQUESTS E.G. *GET INTERFACE DEFINITION

FIG. 3

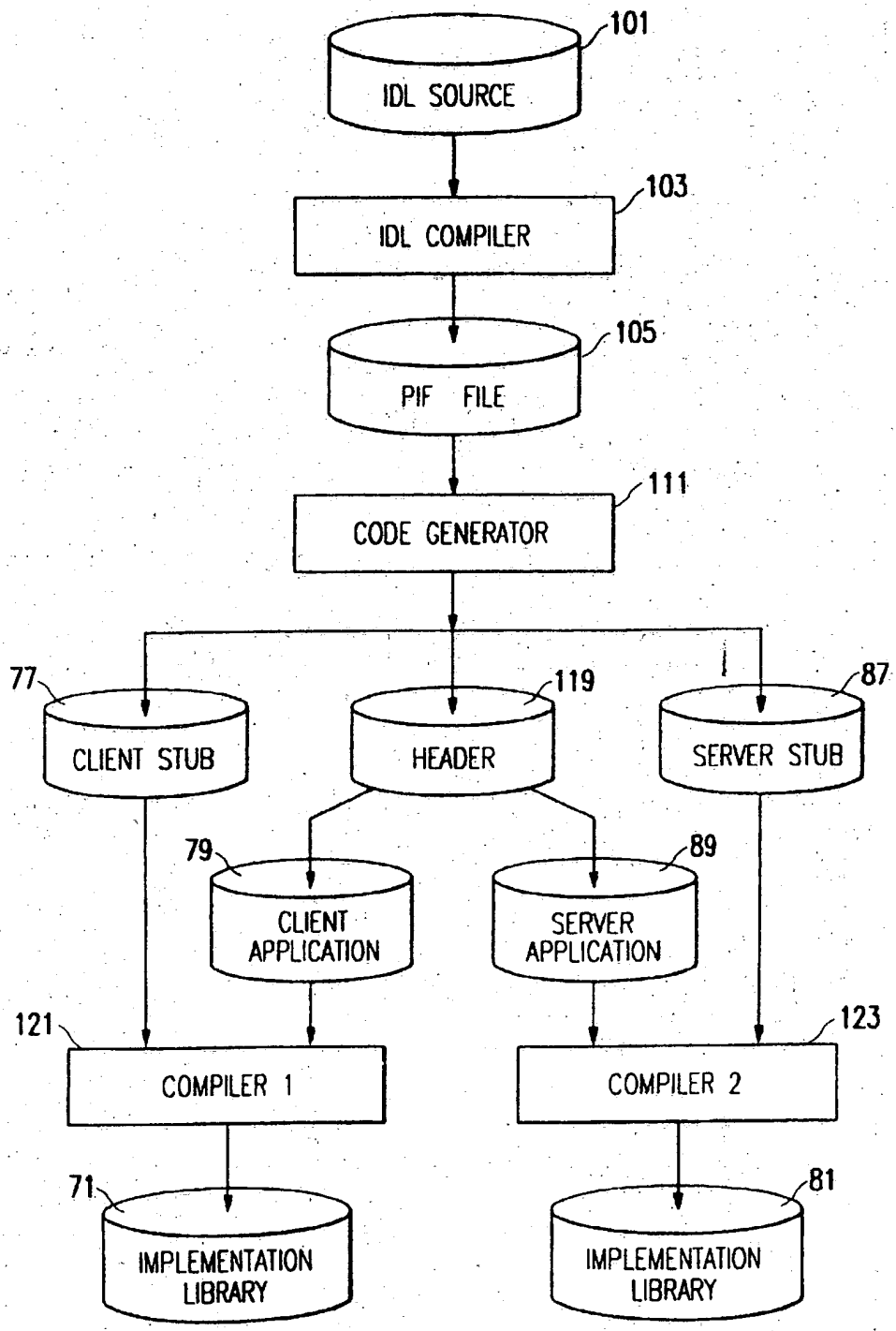

101
IDL SOURCE
103
IDL COMPILER
105
PIF FILE
111
CODE GENERATOR
77
CLIENT STUB
119
HEADER
87
SERVER STUB
79
CLIENT APPLICATION
89
SERVER APPLICATION
121
COMPILER 1
123
COMPILER 2
71
IMPLEMENTATION LIBRARY
81
IMPLEMENTATION LIBRARY

FIG. 4

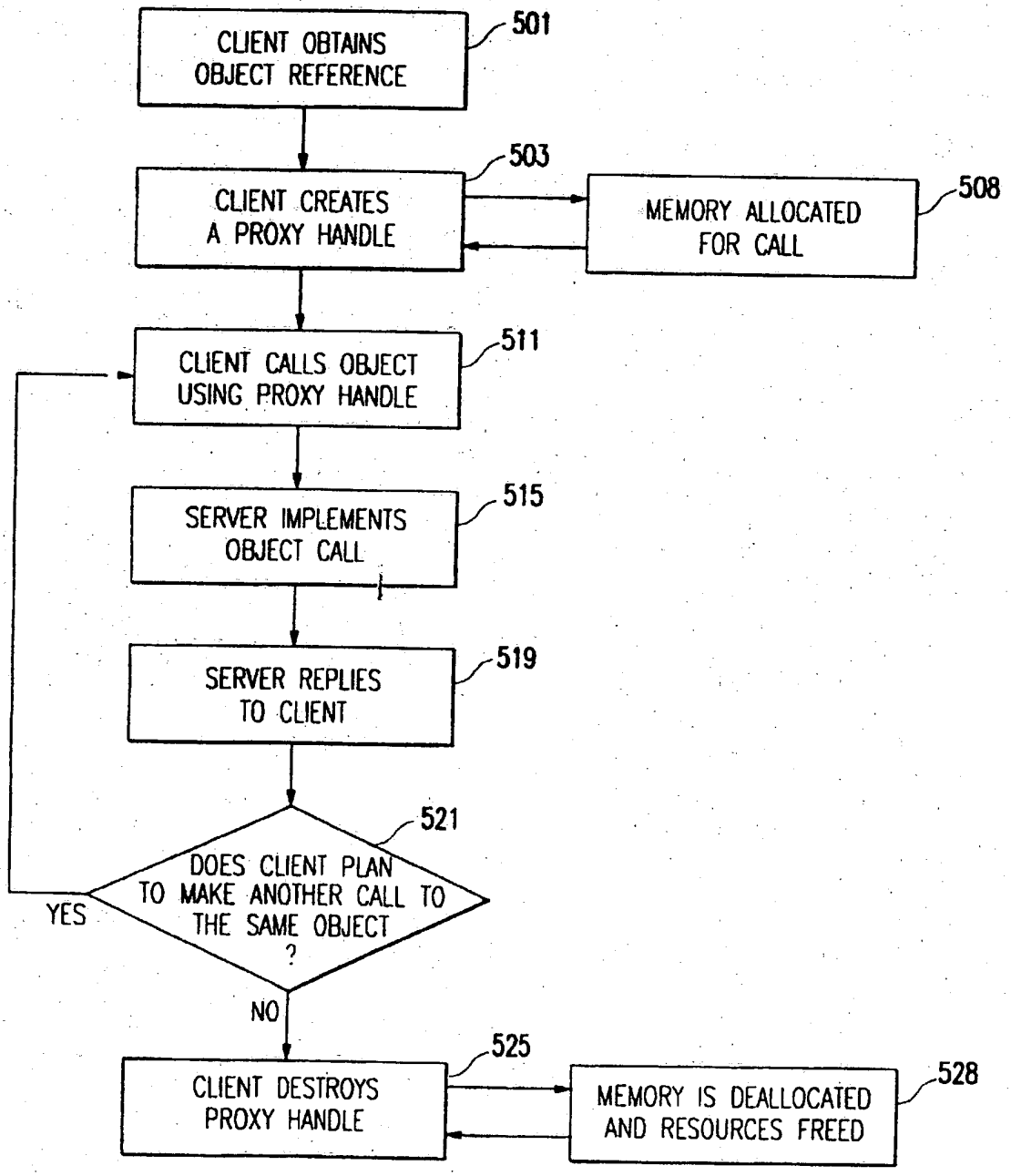

CLIENT OBTAINS OBJECT REFERENCE
501
CLIENT CREATES A PROXY HANDLE
503
MEMORY ALLOCATED FOR CALL
508
CLIENT CALLS OBJECT USING PROXY HANDLE
511
SERVER IMPLEMENTS OBJECT CALL
515
SERVER REPLIES TO CLIENT
519
DOES CLIENT PLAN TO MAKE ANOTHER CALL TO THE SAME OBJECT ?
521
YES
NO
CLIENT DESTROYS PROXY HANDLE
525
MEMORY IS DEALLOCATED AND RESOURCES FREED
528

FIG. 5

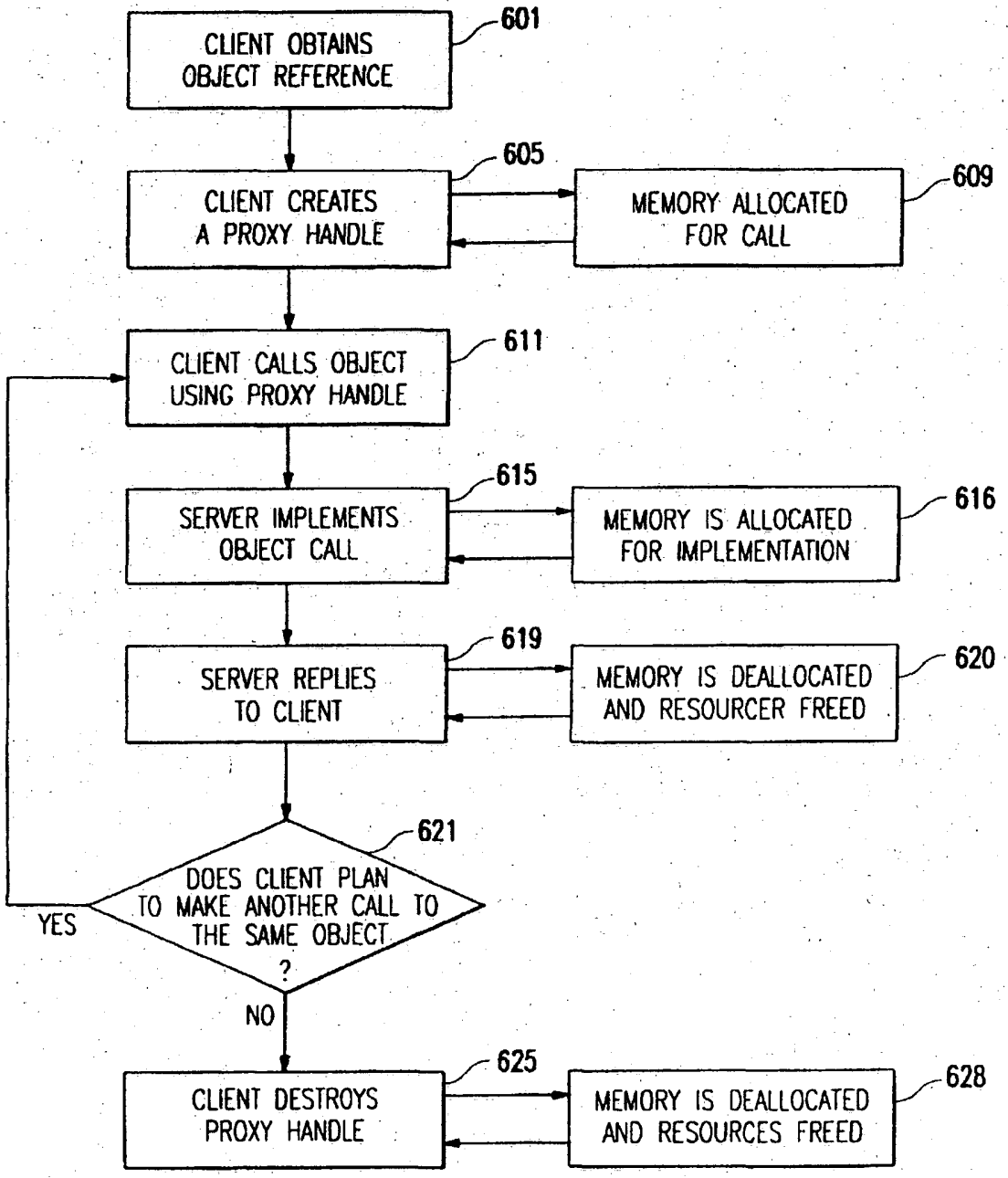

CLIENT OBTAINS OBJECT REFERENCE
601
CLIENT CREATES A PROXY HANDLE
605
MEMORY ALLOCATED FOR CALL
609
CLIENT CALLS OBJECT USING PROXY HANDLE
611
SERVER IMPLEMENTS OBJECT CALL
615
MEMORY IS ALLOCATED FOR IMPLEMENTATION
616
SERVER REPLIES TO CLIENT
619
MEMORY IS DEALLOCATED AND RESOURCER FREED
620
DOES CLIENT PLAN TO MAKE ANOTHER CALL TO THE SAME OBJECT ?
621
YES
NO
CLIENT DESTROYS PROXY HANDLE
625
MEMORY IS DEALLOCATED AND RESOURCES FREED
628

FIG. 6

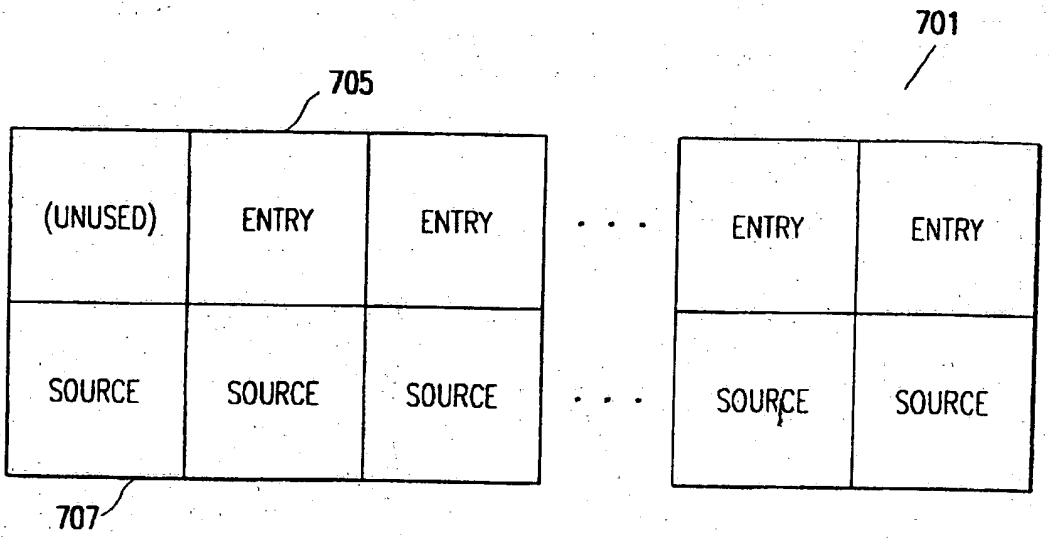

701
705
(UNUSED)
ENTRY
ENTRY
ENTRY
ENTRY
SOURCE
SOURCE
SOURCE
SOURCE
SOURCE
707

FIG. 7

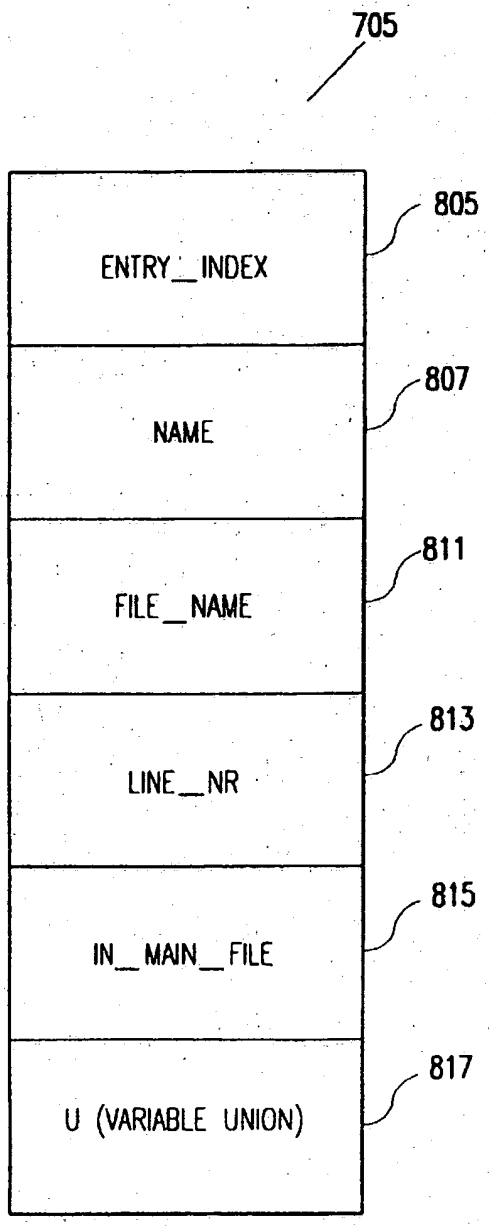

705
ENTRY_INDEX
805
NAME
807
FILE_NAME
811
LINE_NR
813
IN_MAIN_FILE
815
U (VARIABLE UNION)
817

FIG. 8

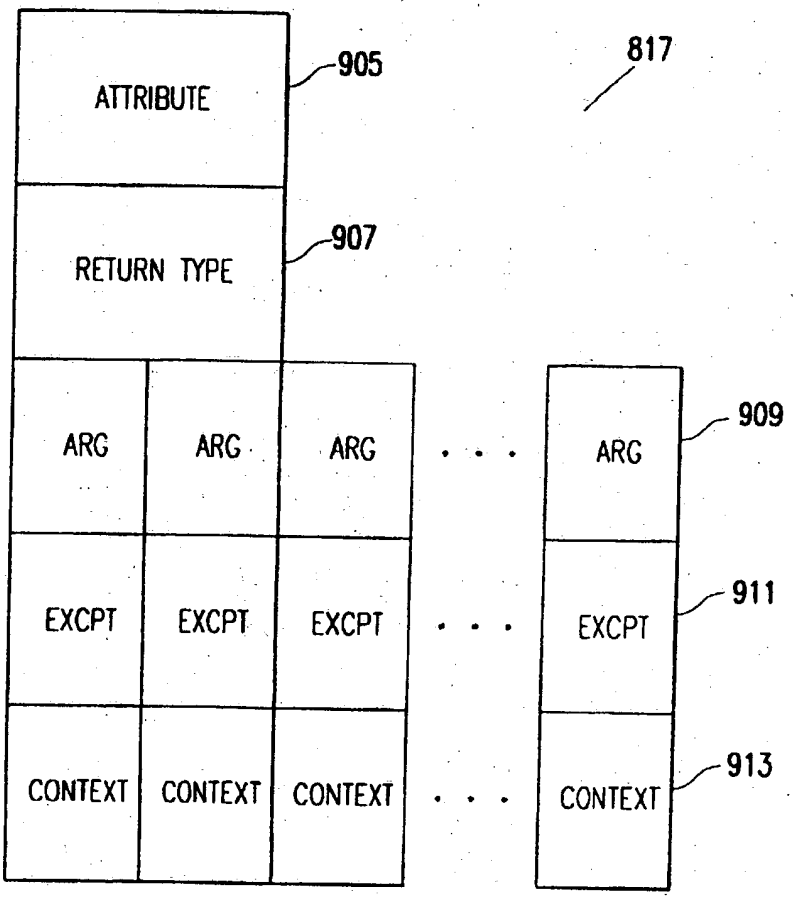

ATTRIBUTE
905
817
RETURN TYPE
907
ARG
ARG
ARG
ARG
909
EXCPT
EXCPT
EXCPT
EXCPT
911
CONTEXT
CONTEXT
CONTEXT
CONTEXT
913

FIG. 9

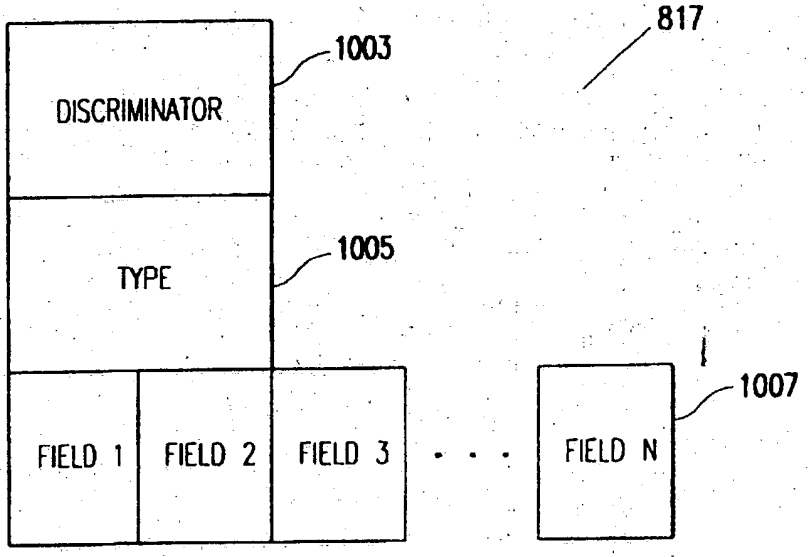

817
1003
DISCRIMINATOR
1005
TYPE
1007
FIELD 1
FIELD 2
FIELD 3
FIELD N

FIG. 10

INTERNATIONAL SEARCH REPORT

International Application No
PCT/US 97/11886

A. CLASSIFICATION OF SUBJECT MATTER
IPC 6 G06F9/46

According to International Patent Classification(IPC) or to both national classification and IPC

B. FIELDS SEARCHED

Minimum documentation searched (classification system followed by classification symbols)
IPC 6 G06F

Documentation searched other than minimum documentation to the extent that such documents are included in the fields searched

Electronic data base consulted during the international search (name of data base and, where practical, search terms used)

C. DOCUMENTS CONSIDERED TO BE RELEVANT

Category *	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
Y	IBM: "SOMObjectsDeveloper Toolkit User Guide, Version 2.1 (Chapter 6)" October 1994 , IBM , US XP002047926 see page 6-20, line 1 - line 16 see page 6-22, line 16 - page 6-24, last line see page 6-27, line 5 - page 6-30, line 22 see page 6-48, line 41 - line 58 see page 6-66, line 35 - page 6-67, line 5 --- -/--	1-4,7-18

[X] Further documents are listed in the continuation of box C.

[X] Patent family members are listed in annex.

* Special categories of cited documents :

"A" document defining the general state of the art which is not considered to be of particular relevance

"E" earlier document but published on or after the international filing date

"L" document which may throw doubts on priority claim(s) or which is cited to establish the publication date of another citation or other special reason (as specified)

"O" document referring to an oral disclosure, use, exhibition or other means

"P" document published prior to the international filing date but later than the priority date claimed

"T" later document published after the international filing date or priority date and not in conflict with the application but cited to understand the principle or theory underlying the invention

"X" document of particular relevance; the claimed invention cannot be considered novel or cannot be considered to involve an inventive step when the document is taken alone

"Y" document of particular relevance; the claimed invention cannot be considered to involve an inventive step when the document is combined with one or more other such documents, such combination being obvious to a person skilled in the art.

"&" document member of the same patent family

Date of the actual completion of the international search	Date of mailing of the international search report
24 November 1997	11/12/1997

Name and mailing address of the ISA	Authorized officer
European Patent Office, P.B. 5818 Patentlaan 2 NL - 2280 HV Rijswijk Tel. (+31-70) 340-2040, Tx. 31 651 epo nl, Fax: (+31-70) 340-3016	Fonderson, A

Form PCT/ISA/210 (second sheet) (July 1992)

INTERNATIONAL SEARCH REPORT

International Application No
PCT/US 97/11886

C.(Continuation) DOCUMENTS CONSIDERED TO BE RELEVANT

Category °	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
Y	JACQUEMOT C ET AL: "COOL: THE CHORUS CORBA COMPLIANT FRAMEWORK" INTELLECTUAL LEVERAGE: DIGEST OF PAPERS OF THE SPRING COMPUTER SOCI INTERNATIONAL CONFERENCE (COMPCON), SAN FRANCISCO, FEB. 28 - MAR. 4, 1994, no. -, 28 February 1994, INSTITUTE OF ELECTRICAL AND ELECTRONICS ENGINEERS, pages 132-141, XP000479388 see page 134, right-hand column, line 41 - page 137, left-hand column, paragraph 1; figure 2 ---	1-4,7-18
A	"DISTRIBUTED OBJECT ACTIVATION AND COMMUNICATION PROTOCOLS" IBM TECHNICAL DISCLOSURE BULLETIN, vol. 37, no. 7, 1 July 1994, pages 539-542, XP002009565 see the whole document -----	1-4,7-18